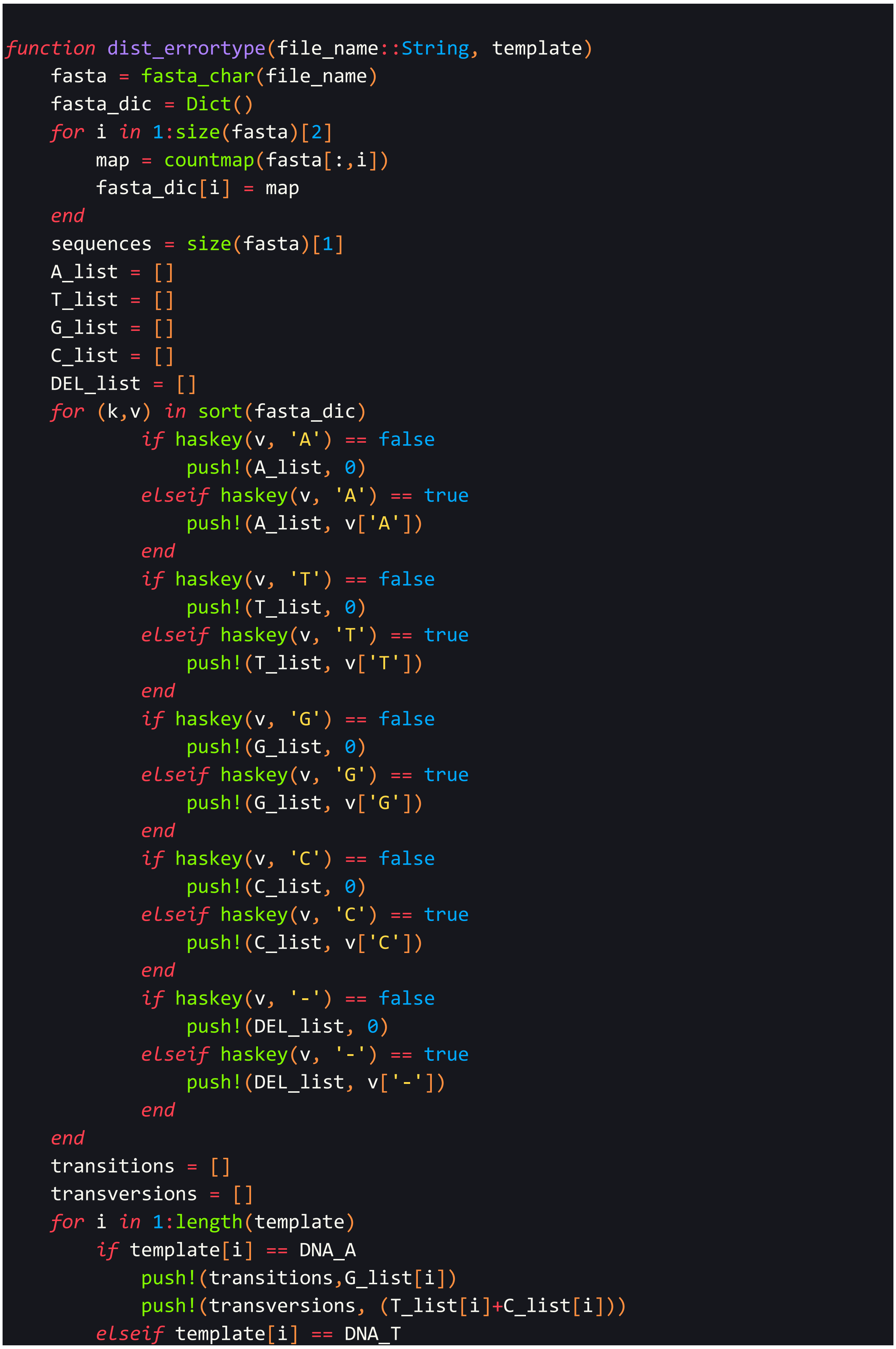
```julia
using FASTX
using BioSequences
using DataFrames
using CSV
using StatsBase
using StatsPlots
using CSV

##############################################################################
######################################

# PART 1
#Extracting sequences with the correct 5' and 3' end priming regions and
trimming to the matching regions

##############################################################################
######################################

# Priming regions and reference sequence
PH_pET23_DA_Biotin3 = dna"AGGGTtAATGCCAGC"
P2_fidelity_inestR1 = dna"CTTGCGGCcacacAG"
Fidelity_ref =
dna"AGGGTtAATGCCAGCGCTTCGTTAATACAGATGTAGGTGTTCCACAGGGTAGCCAGCAGCATATGGTGCAGGGC
GCTGACTTCCGCGTTTCCAGACTTTACGAAACACGGAAACCGAAGACCATTCATGTTGTTGCTCAGGTCGCAGACGTT
TTGCAGCAGCAGTCGCTTCACGTTCGCTCGCGTATCGGTGATTCATTCTGCTAACCAGTAAGGCAACCCCGCCAGCCT
AGCCGGGTCCTCAACGACAGGAGCACGATCATGCGCACCCGTGGCCAGGACCCAACGCTGCCCGAGATCTCGATCCCG
CGAAATTAATACGACTCACTATAGGGAGACCACAACGGTTTCCCTCTAGAAATAATTTTGTTTAACTTTAAGAAGGAG
ATATACCATGGATCCTCTAGAGTCGACCTGCAGGCATGCAAGCTTGCGGCcacacAG"
ref = Fidelity_ref

############# loading FASTA & convert to Strings of Chars ##############
function load_fasta_file(file_name::String)
    seqs = []
    reader = open(FASTA.Reader, file_name)
    for record in reader
        seq = FASTX.FASTA.sequence(record)
        push!(seqs, seq)
    end
    close(reader)
    return seqs
end

############# finding fragments based on ID ##############
function find_and_trim(data)
    seqs = []
    for i in 1:length(data)
        pos1 = findfirst(ExactSearchQuery(PH_pET23_DA_Biotin3), data[i])
        pos2 = findfirst(ExactSearchQuery(P2_fidelity_inestR1), data[i])
        if isnothing(pos1) || isnothing(pos2)
            continue
        end
        trimmed_seq = data[i][pos1.start:pos2.stop]
        push!(seqs,trimmed_seq)
    end

    return seqs
end


# export array as fasta file
function array_to_fasta_file(file_name::String, DATA)
    reader = open(FASTA.Writer, file_name)
        for i in 1:size(DATA)[1]
            seq = DATA[i]
            record = FASTA.Record("Seq$i", seq)
            write(reader, record)
        end
    close(reader)
end

# Running functions
EXO_seqs =load_fasta_file("Galaxy250-EXO-THR.fasta")
DEL_seqs =load_fasta_file("Galaxy249-DEL.fasta")
D12A_seqs =load_fasta_file("Galaxy248-D12A-THR.fasta")
P2NEB_seqs =load_fasta_file("Galaxy251-P2NEB.fasta")

EXO = find_and_trim(EXO_seqs)
DEL = find_and_trim(DEL_seqs)
D12A = find_and_trim(D12A_seqs)
P2NEB = find_and_trim(P2NEB_seqs)

#Output trimmed fasta files
array_to_fasta_file("EXO.fasta", EXO)
array_to_fasta_file("DEL.fasta", DEL)
array_to_fasta_file("D12A.fasta", D12A)

#adding manually the reference sequence as 'ref' to each fasta file before
alignment


##############################################################################
######################################

#PART 2
#Processing multiple sequence alignments
```

```julia
##############################################################################
######################################

#Loading multiple sequence alignments
Dic_EXO = Dict{String, LongSequence}()
reader = FASTA.Reader(open("EXO_MSA.fasta", "r"))
    for record in reader
        Dic_EXO[identifier(record)] = sequence(record)
    end
close(reader)


Dic_D12A = Dict{String, LongSequence}()
reader = FASTA.Reader(open("D12A_MSA.fasta", "r"))
    for record in reader
        Dic_D12A[identifier(record)] = sequence(record)
    end
close(reader)

Dic_DEL = Dict{String, LongSequence}()
reader = FASTA.Reader(open("DEL_MSA.fasta", "r"))
    for record in reader
        Dic_DEL[identifier(record)] = sequence(record)
    end
close(reader)

# extracting reference sequence from alignment
EXO_template = Dic_EXO["ref"]
D12A_template = Dic_D12A["ref"]
DEL_template = Dic_DEL["ref"]

# Calculating entropy per position in the MSA as a measure of diversity
function fasta_char(file_name::String)
    fasta_file = []
    reader = open(FASTA.Reader, file_name)
    for record in reader
        seq = FASTX.FASTA.sequence(record)
        push!(fasta_file, seq)
    end
    close(reader)
    seqs_char = Array{Char, 2}(undef,
length(fasta_file),length(fasta_file[1]))
    for i = 1:length(fasta_file)
           for j = 1:length(fasta_file[i])
               seqs_char[i,j] = fasta_file[i][j]
           end
    end
return seqs_char
end
```

```julia
function probabilities(X)
    counts = countmap(collect(eachrow(X)))
    probs = values(counts)./sum(values(counts))
    return probs
end

function entropy_system(X)
    H = []
    for i in 1:size(X)[2]
        prob = probabilities(X[:,i])
        entropy = (-1.0).*sum(log2.(prob).*prob)
        push!(H, entropy)
    end
    return H
end

# Running functions and exporting output as CSV
EXO_Hx = entropy_system(fasta_char("EXO_MSA.fasta"))
EXO_Hx_df = DataFrame(Reference_seq = collect(EXO_template), Exo_H= EXO_Hx)
CSV.write("EXO_Entropy.csv", EXO_Hx_df)

D12A_Hx = entropy_system(fasta_char("D12A_MSA.fasta"))
D12A_Hx_df = DataFrame(Reference_seq = collect(D12A_template), D12A_H=
D12A_Hx)
CSV.write("D12A_Entropy.csv", D12A_Hx_df)

DEL_Hx = entropy_system(fasta_char("DEL_MSA.fasta"))
DEL_Hx_df = DataFrame(Reference_seq = collect(DEL_template), DEL_H= DEL_Hx)
CSV.write("DEL_Entropy.csv", DEL_Hx_df)

# Entropy spike analysis
exo_spike = countmap(fasta_char("EXO_MSA.fasta")[:,219])
d12a_spike = countmap(fasta_char("D12A_MSA.fasta")[:,231])
del_spike = countmap(fasta_char("DEL_MSA.fasta")[:,245])


# Counting Insertion, deletion and substitution errors
function comparison(counts::Dict{String, Int64}, template::LongSequence,
sequence::LongSequence)
    for i in 1:length(template)
        if template[i] == DNA_Gap && sequence[i] != DNA_Gap
            if i > 1
                if template[i-1] != DNA_Gap
                    counts["insertion"] += 1
                end
            else
                counts["insertion"] += 1
            end
        elseif template[i] != DNA_Gap && sequence[i] == DNA_Gap
            if i > 1
                if sequence[i-1] != DNA_Gap
                    counts["deletion"] += 1
                end
            else
                counts["deletion"] += 1
            end
        elseif template[i] != sequence[i]
            counts["mutation"] += 1
        end
    end
    return counts
end


function pol_fidelity(pol_data::Dict{String,Dict{String,Int64}})
    insertions = []
    deletions = []
    mutations = []
    seq_length = length(ref)
    for (k, v) in pol_data
            ins = pol_data[k]["insertion"]
            dels = pol_data[k]["deletion"]
            muts = pol_data[k]["mutation"]
            push!(insertions, ins)
            push!(deletions, dels)
            push!(mutations, muts)
    end
    total_bases = seq_length*length(collect(keys(pol_data)))
    freq_insertions = sum(insertions)/total_bases
    freq_deletions = sum(deletions)/total_bases
    freq_mutations = sum(mutations)/total_bases
    total = freq_insertions+freq_deletions+freq_mutations
    dict = Dict("insertions/base" => freq_insertions, "deletions/base" =>
freq_deletions,
                "substitution/base" => freq_mutations, "total errors/base" =>
total, "bases sequenced" =>total_bases)
    return dict
end

# Running functions

EXO_counts_per_seq = Dict{String, Dict{String, Int64}}()
for (k, v) in Dic_EXO
    counts = Dict("insertion" => 0, "deletion" => 0, "mutation" => 0)
    comparison(counts, EXO_template, v)
    EXO_counts_per_seq[k] = counts
end
```

```julia
D12A_counts_per_seq = Dict{String, Dict{String, Int64}}()
for (k, v) in Dic_D12A
    counts = Dict("insertion" => 0, "deletion" => 0, "mutation" => 0)
    comparison(counts, D12A_template, v)
    D12A_counts_per_seq[k] = counts
end

DEL_counts_per_seq = Dict{String, Dict{String, Int64}}()
for (k, v) in Dic_DEL
    counts = Dict("insertion" => 0, "deletion" => 0, "mutation" => 0)
    comparison(counts, DEL_template, v)
    DEL_counts_per_seq[k] = counts
end

# Output
pol_fidelity(EXO_counts_per_seq)
pol_fidelity(D12A_counts_per_seq)
pol_fidelity(DEL_counts_per_seq)


# Counting substitution error types (transitions and transversions)

function errortype(error_type::Dict{String, Int64}, template::LongSequence,
sequence::LongSequence)
    for i in 1:length(template)
        if template[i] == DNA_A && sequence[i] == DNA_G
            error_type["A->G"] += 1
        elseif template[i] == DNA_G && sequence[i] == DNA_A
            error_type["G->A"] += 1
        elseif template[i] == DNA_T && sequence[i] == DNA_C
            error_type["T->C"] += 1
        elseif template[i] == DNA_C && sequence[i] == DNA_T
            error_type["C->T"] += 1
        elseif template[i] == DNA_A && sequence[i] == DNA_T
            error_type["A->T"] += 1
        elseif template[i] == DNA_T && sequence[i] == DNA_A
            error_type["T->A"] += 1
        elseif template[i] == DNA_A && sequence[i] == DNA_C
            error_type["A->C"] += 1
        elseif template[i] == DNA_C && sequence[i] == DNA_A
            error_type["C->A"] += 1
        elseif template[i] == DNA_T && sequence[i] == DNA_G
            error_type["T->G"] += 1
        elseif template[i] == DNA_G && sequence[i] == DNA_T
            error_type["G->T"] += 1
        elseif template[i] == DNA_C && sequence[i] == DNA_G
            error_type["C->G"] += 1
        elseif template[i] == DNA_G && sequence[i] == DNA_C
```

```julia
            error_type["G->C"] += 1
        end
    end
    return error_type
end


function error_type_counts(pol_data::Dict{String,Dict{String,Int64}})
    a = []
    b = []
    c = []
    d = []
    e = []
    f = []
    g = []
    h = []
    i = []
    j = []
    k = []
    l = []
    for (ks, vs) in pol_data
            AGs = pol_data[ks]["A->G"]
            GAs = pol_data[ks]["G->A"]
            TCs = pol_data[ks]["T->C"]
            CTs = pol_data[ks]["C->T"]
            ATs = pol_data[ks]["A->T"]
            TAs = pol_data[ks]["T->A"]
            ACs = pol_data[ks]["A->C"]
            CAs = pol_data[ks]["C->A"]
            TGs = pol_data[ks]["T->G"]
            GTs = pol_data[ks]["G->T"]
            CGs = pol_data[ks]["C->G"]
            GCs = pol_data[ks]["G->C"]
            push!(a, AGs)
            push!(b, GAs)
            push!(c, TCs)
            push!(d, CTs)
            push!(e, ATs)
            push!(f, TAs)
            push!(g, ACs)
            push!(h, CAs)
            push!(i, TGs)
            push!(j, GTs)
            push!(k, CGs)
            push!(l, GCs)
    end
    total_base_substitutions = sum(vcat(a,b,c,d,e,f,g,h,i,j,k,l))
    avg_a = 100*sum(a)/total_base_substitutions
    avg_b = 100*sum(b)/total_base_substitutions
    avg_c = 100*sum(c)/total_base_substitutions
    avg_d = 100*sum(d)/total_base_substitutions
    avg_e = 100*sum(e)/total_base_substitutions
    avg_f = 100*sum(f)/total_base_substitutions
    avg_g = 100*sum(g)/total_base_substitutions
    avg_h = 100*sum(h)/total_base_substitutions
    avg_i = 100*sum(i)/total_base_substitutions
    avg_j = 100*sum(j)/total_base_substitutions
    avg_k = 100*sum(k)/total_base_substitutions
    avg_l = 100*sum(l)/total_base_substitutions
    dict = Dict("total"=>total_base_substitutions,"A->G"=>avg_a,"G-
>A"=>avg_b,"T->C"=>avg_c,"C->T"=>avg_d,"A->T"=>avg_e,"T->A"=>avg_f,"A-
>C"=>avg_g,"C->A"=>avg_h,"T->G"=>avg_i,"G->T"=>avg_j,"C->G"=>avg_k,"G-
>C"=>avg_l)
    return dict
end

# Running functions
EXO_errortype = Dict{String, Dict{String, Int64}}()
for (k, v) in Dic_EXO
    error_type = Dict("A->G"=>0,"G->A"=>0,"T->C"=>0,"C->T"=>0,"A->T"=>0,"T-
>A"=>0,"A->C"=>0,"C->A"=>0,"T->G"=>0,"G->T"=>0,"C->G"=>0,"G->C"=>0)
    errortype(error_type, EXO_template, v)
    EXO_errortype[k] = error_type
end

D12A_errortype = Dict{String, Dict{String, Int64}}()
for (k, v) in Dic_D12A
    error_type = Dict("A->G"=>0,"G->A"=>0,"T->C"=>0,"C->T"=>0,"A->T"=>0,"T-
>A"=>0,"A->C"=>0,"C->A"=>0,"T->G"=>0,"G->T"=>0,"C->G"=>0,"G->C"=>0)
    errortype(error_type, D12A_template, v)
    D12A_errortype[k] = error_type
end

DEL_errortype = Dict{String, Dict{String, Int64}}()
for (k, v) in Dic_DEL
    error_type = Dict("A->G"=>0,"G->A"=>0,"T->C"=>0,"C->T"=>0,"A->T"=>0,"T-
>A"=>0,"A->C"=>0,"C->A"=>0,"T->G"=>0,"G->T"=>0,"C->G"=>0,"G->C"=>0)
    errortype(error_type, DEL_template, v)
    DEL_errortype[k] = error_type
end

# Output
EXO_errortype_per = error_type_counts(EXO_errortype)
D12A_errortype_per = error_type_counts(D12A_errortype)
DEL_errortype_per = error_type_counts(DEL_errortype)


# Error type distribution per position
```



```julia
function dist_errortype(file_name::String, template)
    fasta = fasta_char(file_name)
    fasta_dic = Dict()
    for i in 1:size(fasta)[2]
        map = countmap(fasta[:,i])
        fasta_dic[i] = map
    end
    sequences = size(fasta)[1]
    A_list = []
    T_list = []
    G_list = []
    C_list = []
    DEL_list = []
    for (k,v) in sort(fasta_dic)
            if haskey(v, 'A') == false
                push!(A_list, 0)
            elseif haskey(v, 'A') == true
                push!(A_list, v['A'])
            end
            if haskey(v, 'T') == false
                push!(T_list, 0)
            elseif haskey(v, 'T') == true
                push!(T_list, v['T'])
            end
            if haskey(v, 'G') == false
                push!(G_list, 0)
            elseif haskey(v, 'G') == true
                push!(G_list, v['G'])
            end
            if haskey(v, 'C') == false
                push!(C_list, 0)
            elseif haskey(v, 'C') == true
                push!(C_list, v['C'])
            end
            if haskey(v, '-') == false
                push!(DEL_list, 0)
            elseif haskey(v, '-') == true
                push!(DEL_list, v['-'])
            end
    end
    transitions = []
    transversions = []
    for i in 1:length(template)
        if template[i] == DNA_A
            push!(transitions,G_list[i])
            push!(transversions, (T_list[i]+C_list[i]))
        elseif template[i] == DNA_T
```

```julia
            push!(transitions,C_list[i])
            push!(transversions, (A_list[i]+G_list[i]))
        elseif template[i] == DNA_G
            push!(transitions,A_list[i])
            push!(transversions, (T_list[i]+C_list[i]))
        elseif template[i] == DNA_C
            push!(transitions,T_list[i])
            push!(transversions, (A_list[i]+G_list[i]))
        end
    end

    freq_transitions = []
    freq_transversions = []
    for i in 1:length(transitions)
            push!(freq_transitions,(transitions[i]/sequences))
            push!(freq_transversions,(transversions[i]/sequences))
    end
    return DataFrame(Transitions=transitions,Transversions=transversions,
                     Freq_transitions=freq_transitions,Freq_transversions=freq
_transversions)
end


# Running functions
EXO_dist = dist_errortype("EXO_MSA.fasta", EXO_template)
D12A_dist = dist_errortype("D12A_MSA.fasta", D12A_template)
DEL_dist = dist_errortype("DEL_MSA.fasta", DEL_template)

error_map = hcat(EXO_dist,D12A_dist,DEL_dist, makeunique=true)
rename!(error_map,[:Exo_transitions,:Exo_transversions,:Exo_transitions_freq,
:Exo_transversions_Freq,
                    :D12A_transitions,:D12A_transversions,:D12A_transitions_fr
eq, :D12A_transversions_Freq,
                    :DEL_transitions,:DEL_transversions,:DEL_transitions_freq,
:DEL_transversions_Freq,
                    ])

# Output
CSV.write("mutants_error_distribution.csv", error_map)


# Identifying the location and abundance of deletions and insertions
function concat_range(list)
    dats =[]
    for i in 1:length(list)
        dat = collect(list[i])
        append!(dats,dat)
    end
    list = union(dats)
    final_range = []
    vectorized_range = []
    start = 0
    finish = 0
    del_length = 0
    for element = 1:length(list)
        ## Starting new ranges
        if element == 1
            start = list[1]
            finish = list[1]
            del_length += 1
        elseif element > 1 && del_length == 0
            start = list[element]
            finish = list[element]
            del_length += 1
        ## Extending new ranges
        elseif del_length > 0 && list[element-1] == list[element]-1 &&
list[element] != list[end]
            finish = list[element]
            del_length += 1
        ## Closing the ranges
        elseif del_length == 1 && list[element-1] != list[element]-1 &&
list[element] != list[end]
            # Close the previous range
            push!(final_range, [start:finish])
            push!(vectorized_range, [start, del_length])
            # open the new range
            start = list[element]
            finish = list[element]
            del_length = 1
        elseif del_length > 1 && list[element-1] != list[element]-1 &&
list[element] != list[end]
            # Close the previous range
            finish = list[element-1]
            push!(final_range, [start:finish])
            push!(vectorized_range, [start, del_length])
            # Open the new range
            start = list[element]
            finish = list[element]
            del_length = 1
        elseif list[element] == list[end]
            # Close the range
            finish = list[end]
            del_length += 1
            push!(final_range, [start:finish])
            push!(vectorized_range, [start, del_length])
        end
    end
```

```julia
    return reduce(vcat, final_range)
end

function find_dels(Dic, reference)
    df_dels = []
    del_freq = Dict()
    gaps = []
    ref_gaps = countmap(concat_range(findall(r"-",string(reference))))
    for (k,v) in Dic
        seq = string(v)
        positions_seq = concat_range(findall(r"-",seq))
        append!(gaps, positions_seq)
    end
    count_gaps = countmap(gaps)
    unique_ranges = setdiff(collect(keys(count_gaps)),
collect(keys(ref_gaps)))
    for i in 1:length(unique_ranges)
        del_freq[unique_ranges[i]] = count_gaps[unique_ranges[i]]
    end
    for (keys,vals) in del_freq
        len = length(keys)
        range = keys
        freq = vals
        start_del = parse(Int64,split(string(range),":")[1])
        push!(df_dels, (freq,range,start_del,len))
    end
    df_dels = DataFrame([[df_dels[k][kk] for k in 1:length(df_dels)] for kk in
1:length(df_dels[1])], [:Freq, :Range, :Start, :Length])
    return sort(df_dels, :Freq, rev=true)
end

function find_ins(Dic, reference)
    df_ins = []
    ins_freq = Dict()
    gaps = []
    ref_gaps = countmap(concat_range(findall(r"-",string(reference))))
    for (k,v) in Dic
        seq = string(v)
        positions_seq = concat_range(findall(r"-",seq))
        append!(gaps, positions_seq)
    end
    count_gaps = countmap(gaps)
    unique_ranges = intersect(collect(keys(count_gaps)),
collect(keys(ref_gaps)))
    for i in 1:length(unique_ranges)
        ins_freq[unique_ranges[i]] = count_gaps[unique_ranges[i]]
    end
    for (keys,vals) in ins_freq
```

```
        len = length(keys)
        range = keys
        freq = vals
        start_ins = parse(Int64,split(string(range),":")[1])
        push!(df_ins, (freq,range,start_ins,len))
    end
    df_ins = DataFrame([[df_ins[k][kk] for k in 1:length(df_ins)] for kk in
1:length(df_ins[1])], [:Freq, :Range, :Start, :Length])
    return sort(df_ins, :Freq, rev=true)
end

#Running functions
exo_del_map = find_dels(Dic_EXO, EXO_template)
d12a_del_map = find_dels(Dic_D12A, D12A_template)
del_del_map = find_dels(Dic_DEL, DEL_template)

exo_ins_map = find_ins(Dic_EXO, EXO_template)
d12a_ins_map = find_ins(Dic_D12A, D12A_template)
del_ins_map = find_ins(Dic_DEL, DEL_template)

#Output as CSV
CSV.write("exo_del_map.csv", exo_del_map)
CSV.write("d12a_del_map.csv", d12a_del_map)
CSV.write("del_del_map.csv", del_del_map)

CSV.write("exo_ins_map.csv", exo_ins_map)
CSV.write("d12a_ins_map.csv", d12a_ins_map)
CSV.write("del_ins_map.csv", del_ins_map)
```